# A SPARQL distributed query processing engine addressing both vertical and horizontal data partitions

*Un moteur de traitement de requêtes SPARQL distribuées optimisé pour les partitions de données verticales et horizontales*

Abdoul Macina
Université Côte d'Azur, CNRS, I3S, France
macina@i3s.unice.fr

Johan Montagnat
Université Côte d'Azur, CNRS, I3S, France
johan.montagnat@cnrs.fr

Olivier Corby
Université Côte d'Azur, Inria, CNRS, I3S, France
olivier.corby@inria.fr

## ABSTRACT

An increasing number of linked knowledge bases are openly accessible over the Internet. Distributed Query Processing (DQP) techniques enable querying multiple knowledge bases coherently. However, the precise DQP semantics is often overlooked, and query performance issues arise.

In this paper, we propose a DQP engine for distributed RDF graphs, adopting a SPARQL-compliant DQP semantics. We improve performance through heuristics that generate Basic Graph Pattern-based sub-queries designed to maximise the parts of the query processed by the remote endpoints.

We evaluate our DQP engine considering a query set representative of most common SPARQL clauses and different data distribution schemes. Results show a significant reduction of the number of remote queries executed and the query execution time while preserving completeness.

*Un nombre grandissant de bases de connaissances liées sont exposées à travers l'Internet. Le traitement de requêtes distribuées (DQP) permet d'interroger des bases de connaissances multiples simultanément. Cependant, la sémantique DQP précise est souvent négligée, et des problèmes de performance doivent être traités.*

*Dans ce papier, nous proposons un moteur DQP pour l'interrogation de graphs RDF distribués, conforme à la sémantique de SPARQL. Nous en améliorons la performance grâce à des heuristiques qui génèrent des sous-requêtes à partir de schémas de graphes basiques (BGPs) de manière à maximiser la partie de la requête traitée par les serveurs de données distants.*

*Nous évaluons notre moteur DQP à travers un ensemble de reqêtes représentatives de clauses SPARQL les plus répendues et des schémas de distribution des données divers. Les résultats montrent un réduction significative du nombre de requêtes exécutées et du temps de traitement sans altération de la complétude des résultats.*


*BDA* 2016, 15 au 18 Novembre, Poitiers, France.

## Keywords

Distributed Query Processing, SPARQL, Linked data

## 1. INTRODUCTION

The Semantic Web development led to the dissemination of open linked knowledge bases in RDF format[1], distributed world-wide over the Internet. The use of unique Internationalized Resource Identifiers (IRIs) to identify data elements and the sharing of these universal identifiers over multiple data sets create a network of connections between knowledge bases that are operated independently. SPARQL [6] is an expressive language and widely adopted standard for RDF graphs querying. It only partially addresses the problem of querying data distributed over different graphs though, providing a low-level interface to express data distribution-specific queries. The aim of this work is to enable SPARQL querying over multiple RDF knowledge bases transparently. We adopt a user-centric vision, where the result of a SPARQL query applied to a set of graphs should be identical to the result that would be obtained by applying the same query to a virtual graph aggregating all target graphs.

Two main approaches are typically considered to implement querying over distributed data sources [8]: materialisation (also referred to as Extract-Transform-Load) where all data is collected to a single database, thus implementing the virtual data source to be queried, and Distributed Query Processing (DQP), where original queries are transformed, applied to target data sources, and their results integrated. Although materialisation is much simpler to implement, it suffers many drawbacks among which the time and space needed to integrate all data. In this paper, we are more particularly interested in SPARQL DQP for RDF data sources. We investigate (i) the problem of the SPARQL query semantic preservation in a DQP context, which involves query rewriting, and (ii) an optimisation technique to improve query performance, taking into account the way data is partitioned over multiple data sources.

After a study of the state of the art, this paper describes the semantic that we adopt for SPARQL queries in a DQP context in Section 3. Our goal is to preserve SPARQL semantics and expressiveness as much as possible. The DQP process described matches the user-centric vision of a virtual knowledge graph aggregating all data. The problem of query rewriting is outlined and a query decomposition technique that intends to push as much as possible of the

[1]RDF: http://www.w3.org/TR/rdf11-concepts

original query towards the remote sources for maximal filtering of the query results is proposed. In Section 4, we describe our query engine architecture and the algorithms implementing our DQP optimisation strategy. Results are presented in Section 5 based on different types of SPARQL query clauses and different types of data partitioning.

## 2. STATE OF THE ART

### 2.1 RDF data stores distribution context

The RDF W3C recommendation is commonly used for data representation on the Semantic Web. An RDF dataset may be modelled as a graph composed by a set of triples (subject, predicate, object) where subjects and objects are nodes and predicates are the labels of edges linking subjects and objects. Subjects are IRIs (Internationalised Resource Identifiers) or blank nodes, predicates are IRIs and objects are IRIs, blank nodes or literal values. A blank node is a local resource without IRI. RDF triples may belong to named graphs, which correspond to different (possibly overlapping) RDF data sub-graphs. RDF triples without explicit named graph attachment belong to the default graph.

Two main approaches may be considered to query distributed RDF data sets [9]. When the focus is on processing very large graphs, one usualy considers a graph data structure distributed over multiple storage resources. The problem addressed is then the balancing of graph data over multiple storage nodes and the efficiency of the data structure used to split the graph (e.g. [11]). This approach is based on the hypothesis that the graph data can be redistributed so as to improve the performance of the distributed graph querying algorithm used. Conversely, when the focus is on querying legacy linked RDF data stores distributed over the Internet and accessible through SPARQL endpoints, data cannot be repartitioned. The complete graph data structure does not exist as such. Instead, the legacy RDF graphs are implicitly connected through shared IRIs: since IRIs are uniques, nodes using the same IRI in different graphs are identical. This approach is often referred to as federated SPARQL querying (e.g. [14]). Our work falls into this second category.

Legacy data sets stored in independent SPARQL endpoints may typically be *vertically partitioned*, when different kinds of data are stored in disjoint data sources, *horizontally partitioned*, when the same kind of data is stored and distributed in several data sources, or a mix of both, when part of the data is vertically partitioned and part is horizontally partitioned (general case). In RDF graphs, vertical partitioning corresponds to the case where predicates are partitioned among the data stores: a given predicate can only be found in a single data store. Conversely, horizontal partitioning corresponds to the case where all predicates can be founds in several data sources.

### 2.2 SPARQL query language

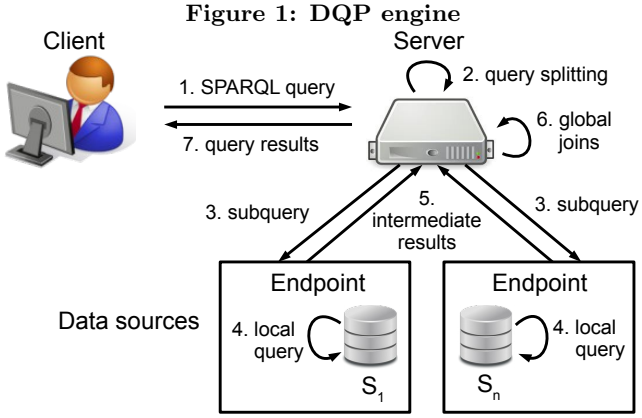
SPARQL is the most common query language used for RDF data. SPARQL queries are composed of two main *clauses*: the first one specifies the query form (SELECT, CONSTRUCT, ASK, or DESCRIBE), while the second one is a WHERE clause specifying a basic graph pattern to be matched by queried graph resources. In the remainder of this paper, we focus on SELECT query form without loss of generality. Most of the DQP processing complexities lies in the queries WHERE clause.

Let us consider a data set representing an organisation composed by some *teams* divided into *groups*. Each group has a number of *members*. Consider the example SPARQL query $Q_1$ below:

```
1 PREFIX ns: <http://examples.fr/team#>
2 SELECT ?name ?members
3 WHERE {
4   ?team   ns:team "SPARKS".
5   ?team   ns:group ?group.
6   ?group  ns:name ?name.
7   ?group  ns:members ?members.
8   }
```

This query is composed of 4 *triple patterns* (TPs) in lines 4 to 7, defining different query patterns for the RDF graph triples, and implicit join operations, when a variable appears in two different TPs.

**Figure 1: DQP engine**




Applying $Q_1$ to a single data source containing all data will return the group name and its number of members for each group in the "SPARKS" team in the database. In this paper however, we will consider data distributed in several sources and accessed through a DQP server as illustrated in Figure 1. The client sends SPARQL queries to the DQP server. The server splits queries to submit sub-queries to remote endpoints. Each endpoint processes locally the sub-queries received and returns partial results. Results are processed on the server side and returned to the client. Join operations may be evaluated locally at an endpoint level or globally at the server level, joining data retrieved from different data sources. In the remainder of this paper, we will refer to *local joins* to refer to endpoint-level joins and *distributed joins* to refer to server-level joins.

SPARQL was natively designed for querying remote data sources. Beyond a graph query and update language, SPARQL defines a protocol for transferring queries towards remote servers (*SPARQL endpoints*) and returning result sets. The SPARQL language also defines a SERVICE clause aiming at applying a sub-query to a specific endpoint identified by its access URI. A SPARQL query can thus be composed by several SERVICE clauses computed on different endpoints, which results are combined together.

A strong limitation of the SERVICE clause in a DQP context is that a query has to be manually decomposed into multiple sub-queries to apply to several data sources. This requires the user to have a precise knowledge of the data partitioning model in the data sources. In the $Q_1$ example

above, if data is split between a data source $S_1$ containing all `ns:team` and `ns:group` relations, and a data source $S_2$ containing all `ns:name` and `ns:members` relations for instance, $Q_1$ will return no results, whether applied to $S_1$ or applied to $S_2$ independently (as it involves joining *groups* which are distributed among $S_1$ and $S_2$). Using SPARQL, a user would have to rewrite the query to decompose it into two SERVICE sub-clauses: a SERVICE clause targeting $S_1$ to retrieve all (`?team`, `?group`) tuples and a SERVICE clause targeting $S_2$ to retrieve all (`?group`, `?name`, `?members`) tuples, before joining all data on *groups*.

SPARQL queries are usually designed to apply to a single data source and designing SERVICE clause-based SPARQL queries to access distributed RDF data may become very cumbersome or even humanly intractable in the case of complex queries rewriting. A technique to implement SPARQL DQP is to automate the rewriting of SPARQL queries into multiple SERVICE clauses. As we will see in Section 3, although this technique is sound it may prove unacceptably inefficient when processing the resulting query and further query optimisation is needed.

## 2.3 Query decomposition strategies

To evaluate SPARQL queries in DQP mode, different strategies may be considered. As discussed above, applying the original query to partial data sources is likely to return a subset of the expected results only since data from different sources need to be joined. A proper DQP strategy will therefore decompose the original query into sub-queries that are relevant for each data source and compute joins later on. Distributed querying typically implies four main steps [10]:

1. *Sources selection*: All data sources are not necessarily containing data that is relevant for a given query. Source selection identifies data sources that are likely to contribute to the result set in order to avoid unnecessary processing.

2. *Query planning* is a query analysis steps aimed at identifying various optimisations that can be applied by transforming the initial query while preserving its semantics, such as reordering query parts.

3. The *query rewriting* step decomposes the original query into a set of sub-queries to be distributed to the remote data servers. It both ensures preservation of the original query semantics and takes into account the performance of the sub-queries to be processed. As a general rule, the more selective the query sent to a remote server, the more efficient the processing.

4. The *Query evaluation* step finally executes the query plan and collects results from all sub-queries generated. Partial results thus aggregated often require further processing (different result sets typically need to be joined) to compute final results.

Sources selection requires information on the sources content. This information is usually acquired prior to query processing, either through data statistics collection on the endpoints, or by probing the endpoints through source sampling queries.

Query planning and query rewriting involve decomposing the original query into source-specific sub-queries so as to retrieve all data that contributes to the final result set. In SPARQL, *Triple-based* evaluation is the finest-grained and simplest query decomposition strategy. It consists in decomposing the query into each of its triple patterns (4 TPs in the case of $Q_1$ for instance), evaluating each TP on the source independently, collecting the matching triples on the server side and joining the results. The obvious drawback of evaluating triples patterns individually is that many triples may be retrieved from the sources and returned to the server, just to be filtered out by the join operations. If this strategy is returning correct results (all potentially matching triples are necessarily returned), it is often inefficient. A simple optimisation consists in computing nested loop joins for triple patterns with join conditions. This limits the number of triples returned to the server for post-processing, but at the cost of many remote queries generation.

Conversely, the *BGP-based* strategy consists in evaluating Basic Graph Patterns (BGPs) sub-queries. BGPs are groups of triple patterns. Their evaluation potentially involves processing joins at the endpoint level (local joins), which is more efficient than retrieving all possible triples at the server level before computing joins (distributed joins). The use of SPARQL SERVICE clauses is a typical way of sending BGPs for evaluation on remote endpoints. However, as discussed before, a BGP could return a subset of the expected results only, due to the need to compute joins between data distributed in different data stores. In the case of vertically partitioned data sets, a whole partition is contained in a single server, to which a global BGP can be sent through a SERVICE clause. In case of horizontally partitioned data sets, BGPs can still be computed but different BGPs may be needed depending on the endpoints content.

To preserve the semantics of the original query, it is mandatory to carefully split it into sub-queries that do not cause results losses. The larger BGPs can be evaluated independently in each data source, the more efficient the query. Yet, the data distribution scheme needs to be taken into account to ensure that a BGP does not filter results too much for complete query processing. An *hybrid BGP-Triple* evaluation strategy computes the largest BPGs usable without alterating the results. It mixes the evaluation of BGPs and TPs at the endpoints level to collect a super-set of the final results over which remaining join operations can be computed at the server level. The difficult part of hybrid strategies is to estimate the most efficient BPGs to process for each data source and the combinations of TPs that complete the query.

## 2.4 Main federated SPARQL DQP engines

Naive DQP implementations may lead to a tremendous number of remote queries and generate large partial result sets. The optimization of DQP is therefore critical and many related work primarily address the problem of query performance. This is the case for several DQP engines designed to query RDF data stores with SPARQL, such as DARQ [13], FedX [14], SPLENDID [7], and ANAPSID [1]. Their data sources selection and query decomposition strategies are described below.

### 2.4.1 Sources selection

Most existing engines rely on prior information on sources content for sources selection. DARQ relies on *service descriptions* to identify relevant sources. The descriptors provide information on predicate capabilities and statistics in-

formation on triples hosted in each source. However, unbound predicates (*i.e.* variables as predicates) are not handled by DARQ. SPLENDID *index manager* uses statistical data on sources expressed through the Vocabulary of Interlinked Data (VoID [2]), to build an index of predicates and their cardinality. This index is computed through SPARQL ASK queries for triple patterns with bound variables and by assigning all sources to triple patterns with unbound predicates. ANAPSID source selection is based on a *catalog* of predicates, concepts capabilities and statistics. The statistics are updated dynamically by the query engine during queries evaluation.

Rich prior information on data sources content is useful to implement elaborated source selection strategies. However, it requires specific instrumentation of data sources, and therefore do not apply to regular SPARQL endpoints. FedX also performs source selection optimisation but unlike the previous approaches, without prior knowledge on sources. Only SPARQL ASK queries are sent to identify the relevant sources for query predicates. Similarly, our DQP engine is based on SPARQL ASK queries for sources statistics collection.

#### 2.4.2 *Query rewriting*

BGP-based query evaluation is implemented by all federated SPARQL query engines listed above to improve performance. DARQ also add filters to BGP-based sub-queries for more selectivity. However, in all these approaches, BGP generation is only considered for triples patterns relevant to a single source (vertically partitioned data). Regarding SPARQL language expressiveness, only ANAPSID can cope with the SPARQL 1.1 standard. FedX, SPLENDID and DARQ support SPARQL 1.0, which does not include the SPARQL SERVICE clause. The hybrid strategy described in this paper can generate BGPs for both vertical and horizontal data partitions. The heuristic implemented aims at creating the largest possible BGPs and applying as much as possible filtering at the endpoints level to minimize the number of queries generated, and the number of partial results collected for post processing at the DQP server level.

## 3. DISTRIBUTING SPARQL QUERIES

In this work, we more specifically address the query rewriting step of the DQP process. The objective is to develop a SPARQL DQP engine that (i) is fully compatible with the SPARQL 1.1 standard while preserving performance and (ii) can adapt to the existing data partitioning scheme.

DQP query performance is a major issue. As outlined in Section 2.4, it is often dealt with in the litterature by only considering a subset of the SPARQL standard, making assumptions on the data distribution or considering instrumented endpoints. To remain user-centric, our approach does not make any constraining assumption on the kind of request applied (full SPARQL compliance), on the data partionning scheme (endpoints are pre-existing and no attempt is made to redistribute the data), nor on the endpoints capability (standard third party SPARQL endpoints can be queried).

By giving language expressiveness the highest priority, we ensure that users will not be limited by the query language and that results returned will be complete with regards to the existing standard. To alleviate the resulting performance problem, we propose a query rewriting heuristic that tackles all kinds of data partitioning schemes (vertical and/or horizontal), using only limited knowledge on endpoints content, that can be inferred through regular SPARQL queries.

Since automated query rewriting to adapt a SPARQL query to data distributed over different sources is not part of the SPARQL standard, a clear semantics for SPARQL DQP first needs to be defined. It should be noted that none of the approaches cited above investigates the DQP semantics.

### 3.1 Distributed SPARQL query semantics

Most works on SPARQL DQP make the assumption that users expect the SPARQL engine to return the same result whether processing a single or multiple data sources. Intuitively, this corresponds to evaluating the original SPARQL query against an aggregated data graph, defined as the set union of all data source triples lists. Implementing this semantics is not straightforward though. Moreover, there are particularities of RDF graphs to be considered to define a precise semantics for the evaluation of SPARQL queries on multiple data sources: *named graphs*, *data replication*, *blank nodes* and *redundant results*.

**Named graphs:** Named graphs are sub-graphs in an original RDF graph, identified by an IRI. The RDF standard does not anticipate data distributed over multiple data stores. As a consequence, the fact that the same IRI for named graphs appears in different RDF stores does not imply anything on the relations between these named graphs. In the DQP context, we consider that data associated to the same named graph IRI belongs to a single named graph, even if it is distributed in different sources. This is coherent with the idea of aggregating all data sources into a single virtual graph. In addition, IRI are usually specific enough to prevent most accidental collisions between names.

**Data replication:** In a distributed context, it is common to replicate data items over several data servers to improve availability. RDF triples may thus be replicated in different servers. When evaluating a query, an RDF triple should be accounted for only one time, even if it is replicated over different servers. This is coherent with the aggregated graph view, which cannot contain duplicated triples.

**Blank nodes:** By definition, blank nodes are locally scoped nodes which identifier (if any) is not an IRI. Hence identifiers of two blank nodes from two different data stores may collide. In the DQP context, blank nodes identifier collisions may occur either for unrelated nodes from two data sources that accidentally share the same identifier, or because a blank node was replicated over different stores. Since it is not possible to discriminate between these two cases automatically, we make the assumption that blank nodes between different sources always differ. If triples containing blank nodes need to be replicated, care should be taken to assign IRIs to these nodes before duplication (a process known as skolemization).

**Redundant results:** Finally, SPARQL can return some results multiple times in a result multiset, if same values match several graph patterns in the original query. In a distributed context, data replication is also likely to lead to redundant results if the same triple is accounted for several time in a query. However, this redundancy in results is only a side effect of data duplication that would disappear if data was aggregated in a single virtual graph. The query engine therefore needs to detect which redundant results are legitimate (products of the SPARQL evaluation) and which ones

are side effects and should be filtered out.

## 3.2 Hybrid query rewriting

We use the following formalism to compute BGPs in our hybrid query evaluation strategy: the $G\bowtie$ operator describes all join operations (distributed and local) when evaluating a SPARQL query, the $L\bowtie$ operator corresponds to local joins computed by endpoints and the $D\bowtie$ operator corresponds to distributed joins computed by the DQP server.

Let TP $= \{tp_1, \ldots, tp_n\}$ be the set of triple patterns from the WHERE clause of a SPARQL query and $S_1 \ldots S_m$ be a set of RDF data sources interfaced through SPARQL endpoints:

- $G\bowtie\ (tp_1, \ldots, tp_n)$ represents the join operation of all triples matching TP triple patterns from a virtual data set aggregating all data sources (SPARQL query computed over the virtual RDF graph, complying to the semantics described in Section 3.1):
  $G\bowtie\ (tp_1, \ldots, tp_n) = \{tp_1, \ldots, tp_n\}$ evaluatedin $\{S_1 \cup S_2 \cup \ldots \cup\ S_m\}$
- $L\bowtie\ (tp_1, \ldots, tp_n)$ represents the union of results from the local join of all triples matching TP triple patterns in each data source (SPARQL *BGP* operator processed in each source):
  $L\bowtie\ (tp_1, \ldots, tp_n) = (\{tp_1, \ldots, tp_n\}\ evaluated\ in\ S_1) \cup (\{tp_1, \ldots, tp_n\}\ in\ S_2) \cup \ldots (\{tp_1, \ldots, tp_n\}\ in\ S_m)$
  Note that $L\bowtie\ (tp_1, \ldots, tp_n)$ is included in $G\bowtie\ (tp_1, \ldots, tp_n)$.
- $D\bowtie\ (tp_1, \ldots, tp_n)$ represents the distributed join operations of all triples matching TP triple patterns (joins results from at least two sources):
  $D\bowtie\ (tp_1, \ldots, tp_n) = G\bowtie\ (tp_1, \ldots, tp_n) \setminus L\bowtie\ (tp_1, \ldots, tp_n)$
  and
  $G\bowtie\ (tp_1, \ldots, tp_n) = L\bowtie\ (tp_1, \ldots, tp_n) \cup D\bowtie\ (tp_1, \ldots, tp_n)$

To illustrate these operators, let us consider the execution of query $Q_1$ introduced in Section 2.2 (TP = {?team `ns:team` "SPARKS", ?team `ns:group` ?group, ?group `ns:name` ?name, ?group `ns:members` ?members}) over the two distributed data sources $S_1$ and $S_2$ described below:

| $S_1$ | $S_2$ |
|---|---|
| t1 ns:team "SPARKS" | t1 ns:team "SPARKS" |
| t1 ns:group g1 | t1 ns:group g2 |
| g1 ns:name "Modalis" | g2 ns:name "Wimmics" |
| g1 ns:members 12 | g2 ns:members 9 |
| t1 ns:group g3 | g3 ns:name "MinD" |
| g3 ns:members 7 | |

The 3 join operators defined above produce the following result sets when applied to the $S_1$ and $S_2$ distributed data sources:

| Operators | Results retrieved |
|---|---|
| $L\bowtie\ (TP)$ | { t1, "SPARKS", g1, "Modalis", 12}<br>{ t1, "SPARKS", g2, "Wimmics", 9} |
| $D\bowtie\ (TP)$ | { t1, "SPARKS", g3, "MinD", 7} |
| $G\bowtie\ (TP)$ | { t1, "SPARKS", g1, "Modalis", 12}<br>{ t1, "SPARKS", g2, "Wimmics", 9}<br>{ t1, "SPARKS", g3, "MinD", 7} |

Indeed, $L\bowtie\ (TP)$ computes a join of all 4 triple patterns in TP in each data sources, producing one 4-items tuple for each of them (group named "Modalis" in $S_1$ and group named "Wimmics" in $S_2$). However, triples associated to the group named "MinD" are distributed between the two data sources and they do not match the $L\bowtie$ local join. Conversely, $D\bowtie\ (TP)$ only accounts for distributed triple joins and therefore retrieves the remaining group. Finally, $G\bowtie\ (TP)$ is the complete result set, resulting from the union of $L\bowtie\ (TP)$ and $D\bowtie\ (TP)$ results.

### 3.2.1 Global join decomposition

The $G\bowtie$ operator is associative and commutative [3, 12]:
$\forall k < n$, $G\bowtie\ (tp_1, \ldots, tp_n)$
$= G\bowtie\ (tp_1, \ldots, tp_k)\ .\ G\bowtie\ (tp_{k+1}, \ldots, tp_n)$
$= G\bowtie\ (tp_{k+1}, \ldots, tp_n)\ .\ G\bowtie\ (tp_1, \ldots, tp_k)$
where " . " represents the binary operator for joins. Consequently, $\forall k < n$, $G\bowtie\ (tp_1, \ldots, tp_n) =$
$(L\bowtie\ (tp_1, \ldots, tp_k) \cup D\bowtie\ (tp_1, \ldots, tp_k))\ .$
$(L\bowtie\ (tp_{k+1}, \ldots, tp_n) \cup D\bowtie\ (tp_{k+1}, \ldots, tp_n))$
This equation can be used to compute the global join operation through a combination of local and distributed joins. Two main cases can be considered. In the case where a source does not contain triples matching all triple patterns in TP, $L\bowtie\ (tp_1 \ldots tp_n)$ is empty and $G\bowtie\ (tp1, \ldots, tp_n) \equiv D\bowtie\ (tp_1, \ldots, tp_n)$. But in the case where a source does not contain triples matching all TP patterns, the $G\bowtie$ operator can be computed as a join of several independent partial joins according to the distribution of triple patterns over sources. The partial joins can be computed as a union of partial $L\bowtie$ and $D\bowtie$.

### 3.2.2 Hybrid evaluation strategy

Our hybrid evaluation strategy is based on the idea of maximising the part of the query processed by the remote endpoints. It therefore pushes as much as possible SPARQL FILTER clauses into the sub-queries generated to improve partial results filtering at the source. It also exploits the local and distributed join operators composition properties shown above since the partial $L\bowtie$ operators size correspond to the evaluation of the largest possible BGPs at the endpoints level. However, this maximisation is constrained by the fact that TPs should be decomposed in disjoint subsets of triple patterns. The hybrid evaluation strategy requires prior knowledge on the data sources partitioning scheme to decide on the best triple patterns partitionning strategy.

### 3.2.3 Local joins generation strategy

Triple patterns which predicates are only available in vertical partitions are used to compose BGPs (one BGP per vertical partition) that can be evaluated independently. Among the remaining triple patterns, two criteria are considered with the aim of creating the largest possible BGPs to maximize the computation delegated to each endpoint:

1. Two triple patterns may be grouped into a BGP only when they contain common variables and therefore they represent inner-joins. Indeed in SPARQL, joining triple patterns without common variable corresponds to computing a Cartesian product between the triples matching each pattern. There is no interest in computing BGPs to reduce the number of results retrieved in this case.

2. The distribution of predicates inside horizontally partitioned data sources needs to be taken into account

in order to generate the disjoint subsets of triple patterns to use in local joins. Let PH be the set of predicates that are horizontally partitioned among the data sources, we distinguish two cases for each data source:

- Global distribution: all predicates from PH are present in a source (triples matching all triple patterns may be found in this source). If there are common variables between all triple patterns, a global $L\bowtie$ (containing all shared triple patterns) will be generated for this source to handle local joins. Otherwise Triple-based evaluation is used.
- Partial distribution: predicates from PH are partially present in sources (only triples matching a subset of all triple patterns may be found in these sources). A partial $L\bowtie$ is generated for each source using the largest possible number of triple patterns with common variables. The remaining triple patterns (without common variables) will be evaluated through the Triple-based approach. Each triple pattern should be applied to the target sources only once. Therefore, triple patterns which are already consumed in distributed and local joins are excluded from the $L\bowtie$ subsequently generated.

#### 3.2.4 Distributed joins generation strategy

$D\bowtie$ operators aim at completing $L\bowtie$ query results by only processing distributed joins for horizontal partitions (there is no distributed join for vertical partitions). Thus, each $L\bowtie$ generated has a corresponding $D\bowtie$. For the $L\bowtie$ generation strategy, we distinguish between global $D\bowtie$, when the distribution of predicates is global in a given data source, and partial $D\bowtie$ when it is not.

Unlike the $L\bowtie$ operators, the evaluation of $D\bowtie$ operators computes query results built with at least two distributed sources. To avoid redundancy with query results already handled by $L\bowtie$, $D\bowtie$ operators need to ensure that the query results found are not local to a source (i.e only built with intermediate results from that source). This is implemented through a pruning algorithm that prevents retrieving the last intermediate results of a distributed query from a given data source when all the previous ones came from this same source.

# 4. DISTRIBUTED QUERY EXECUTION

## 4.1 KGRAM SPARQL engine

The *Knowledge Graph Abstract Machine* (KGRAM) [4] is an implementation of SPARQL 1.1. It is an open source project structured to be easily extensible. Its code is available from github[2]. Among other extension points, KGRAM defines a data producer interface abstracting the data sources connected to the SPARQL server. Data producers may be traditional SPARQL endpoints, alternative database systems, or technical proxies used to connected various data sources.

In an earlier work, we leveraged the data producer interface to implement a *meta-producer* connecting to several data sources concurrently [5].

[2]KGRAM github repository: https://github.com/Wimmics/corese

## 4.2 Hybrid query evaluation strategy

In this work, the KGRAM DQP module has been revised to implement the hybrid optimisation strategy described in Section 3.2.2. In the warmup phase, the module queries remote data sources with SPARQL ASK queries for predicates contained in each of them, to determine the data partitioning scheme. At query execution time, the module computes the sub-queries BGPs and FILTER clauses.

#### 4.2.1 Warmup step

The hybrid query evaluation algorithm makes use of 3 indexes:

- *idxTPSources* associates to each triple pattern a set of data sources hosting candidate RDF triples. It is used to determine which part of the data is horizontally and vertically partitioned.
- *idxSourceTPs* associates to each data source, triple patterns which are exclusive to it. It is used to determine if horizontal data fragmentation is total or partial.
- *idxTPVariables* associates to each triple pattern a set of variables used in this pattern. This index is used to check if several triple patterns have common variables.

From this information, the algorithm rewrites the original query into a set of sub-queries preserving the query semantics. The sub-queries are then evaluated on the remote endpoints and their results joined by the DQP server.

**Data:** idxTPSources, queryExpr
**Result:** the set of SPARQL query results
**foreach** *triplePattern* ∈ *queryExpr* **do**
  **if** *( idxTPSources.get(triplePattern).size() > 1 )* **then**
    **return**
    *evaluate*(*createL⋈UnionD⋈* (*queryExpr*));
  **end**
**end**
**return** *serviceClause*(*createL⋈* (*queryExpr*));

where:

**serviceClause**: generates a SPARQL query with a SERVICE clause to send this BGP to a specific source.

**createL⋈UnionD⋈** (Algorithm 2): joins decomposition.

**evaluate**: handles evaluation as explained in section 4.2.3.

**Algorithm 1:** determines the joins decomposition strategy from a set of triple patterns and the data partitioning scheme.

#### 4.2.2 Query rewriting step

Algorithm 1 describes how BGP expressions are rewritten as unions of $L\bowtie$ and $D\bowtie$ operators, if data is horizontally partitioned, or as a unique $L\bowtie$ (for each source) if data is vertically partitioned. The input for this algorithm is a *queryExpr* expression containing the list of triple patterns and the FILTER clauses from the original SPARQL query.

```
Data: idxSourceTPs, idxTPVariables, queryExpr,
      sourceSet
Result: exprResult rewritten query
if (globalDistribution(sourceSet, queryExpr,
 idxSourceTPs)) then
    if (commonVariables(queryExpr, idxTPVariables))
     then
        return union(createL⋈ (queryExpr),
                     createD⋈ (queryExpr));
    else no common variable
        return queryExpr;
    end
else partial distribution
    sourceSet.sortByNumberOfPredicates();
    exprResult ← ∅;
    BGP_TP_List ← ∅;
    foreach s ∈ sourceSet do
        tps ← idxSourceTPs.get(s);
        if (commonVariables (tps, idxTPVariables))
         then
            ctps = cleanBGP(tps);
            BGP_TP_List.add(ctps);
            exprResult ← join(exprResult,
                union(createL⋈ (ctps),
             createD⋈ (ctps)));
        end
    end
    free_TP_List ←
     set_substract(queryExpr, BGP_TP_List);
    return
     join(exprResult, createD⋈ (free_TP_List));
end
```

where:

**globalDistribution**: returns *true* if all sources contain triple candidates for all triple patterns in BGPExpr.

**commonVariables**: returns *true* if a set of patterns share variables (and therefore describe a join operation).

**createL⋈**: creates a BGP expression from a set of triple patterns forming a BGP and add applicable filters from the original query.

**createD⋈**: creates a triple-based join expression and add applicable filters from the original query.

**cleanBGP**: deletes the triple patterns already handled by another $L\bowtie$ or $D\bowtie$ from a set of triple patterns to keep $L\bowtie$ operators or $D\bowtie$ operators disjoint.

**Algorithm 2:** (**createL⋈UnionD⋈**) decomposes joins in a union of local and distributed joins.

### 4.2.3 Query evaluation step

Algorithm 3 executes the local and distributed join operations for each triple pattern in the query.

```
evaluate:
resultTriples ← ∅;
foreach expr ∈ exprResult do
    switch expr.getType() do
        case UNION do
            resultsTriples ← join(resultsTriples,
                evaluate(expr.leftOp(), expr.rightOp());
        end
        case D⋈ do
            resultsTriples ← join(resultsTriples,
                evaluateD⋈ (expr.getTriplePatterns());
        end
        case L⋈ do
            resultsTriples ← join(resultsTriples,
                evaluateL⋈ (expr.getTriplePatterns());
        end
    end
end
return resultTriples;
```

**evaluateL⋈**: writes a query with all triple patterns in the expression and sends it to sources containing all edges related to these triple patterns.

**evaluateD⋈** (Algorithm 4): evaluates a set of triple patterns in a nested loop join way by writing a query for each triple pattern, and avoids redundancy by pruning.

**Algorithm 3:** (**createL⋈UnionD⋈**) decomposes joins in a union of local and distributed joins.

## 5. EVALUATION

The evaluation of our KGRAM-DQP implementation addresses both the completeness of the results (with a minimal number of sub-queries processing), and query execution performance. The experiments proposed in this section aim at demonstrating the completeness of the results independently from the data partitioning scheme and the kind of SPARQL SELECT query executed. They are based on the querying of two linked RDF datasets, and a set of representative SPARQL queries covering most common SPARQL clauses. In each case, performance is measured in terms of number of sub-queries generated and computation time. All experiments are ran locally on a single dedicated quad-core laptop (Dell Latitude E6430 running Linux Ubuntu 14.04, 2.7 GHz Intel CPU i7-3740QM, 8 GB RAM) running all SPARQL endpoints and the KGRAM query engine, thus preventing any impact from the network load on performance measurements. To further alleviate any problem related to execution time variations that cannot be controlled in a multi-core multi-threaded execution environment, each experiment is re-executed 6 times and computation times averaged.

### 5.1 Experimental set up

The two experimental data sets used are French geographic[3]

[3]INSEE geographic data set: http://rdf.insee.fr/geo/cog-2014.ttl.zip

**Data:** idxSourceTPs, idxTPVariables, triplePatterns, sourceSet
**Result:** Results the set of SPARQL query results.
**foreach** $tp \in triplePatterns$ **do**
  **foreach** $s \in sourceSet$ **do**
    **if** $(tp \in idxSourceTPs(s)$ ;
    &;
    $tp.isNotLastPattern())$ **then**
      $Results \leftarrow process(Results, tp, s)$;
    **else** last edge to handle the pruning
      **if** $(Results.sameSource())$ **then**
        **if** $(Results.sources.notContains(s))$ **then**
          $Results \leftarrow process(Results, tp, s)$;
        **else** nothing to do because already handled by the equivalent $L\bowtie$
        **end**
      **else**
        $Results \leftarrow process(Results, tp, s)$;
      **end**
    **end**
  **end**
**end**
**return** *Results*;

where:

**process:**
**if** $(Results.isEmpty())$ **then**
  $Results \leftarrow query(tp)$;
  $bookKeeping.add(Results, s)$;
**else**
  $tmp \leftarrow query(tp)$;
  $bookKeeping.add(tmp, s)$;
  $Results \leftarrow join(Results, tmp)$;
**end**
**return** *Results*;

**bookKeeping**: handles the history of results to determine if all results came from the same source in order to avoid redundancy.

**Algorithm 4:** (**evaluateD⋈**): evaluates all triple patterns of $D\bowtie$ independently and avoids redundancy by pruning.

and demographic[4] RDF graphs published openly by the National Institute of Statistical and Economical Studies (INSEE). Both contain linked data on the geographical repartition of population on the French administrative territory decompositions.

To reproduce a context of both vertical and horizontal partitioning of data, the demographic data set is vertically partitioned (source $S_1$) and the geographic data set is horizontally partitioned in three test cases:

- $P_1$ partitioning (data duplication): the geographic data is duplicated into two distributed sources ($S_2$ and $S_3$), each containing a complete copy of the data. This test case aims at evaluating the handling of redundant results by each evaluation method.
- $P_2$ partitioning (global distribution of predicates): the geographic data is partitioned into two sources ($S_4$ and $S_5$), each containing all predicates related to this data.
- $P_3$ partitioning (partial distribution of predicates): the geographic data is partitioned into three sources ($S_6$, $S_7$ and $S_8$), each containing a subset of predicates related to this data.

The first partitioning aims at testing the handling of redundant results by all evaluation methods, and the two last ones aim at testing the global and partial distribution strategies introduced in Section 3.2.3. The data partitions are summarized in Table 1.

Six evaluation queries (shown below) were selected as a representative set of SPARQL queries covering the most common clauses: the first query is made of a simple SELECT clause, the second one introduces a UNION, the third one a MINUS, the fourth one FILTERs, the fifth one an OPTIONAL clause, and the last one is a combination of all clauses. In all evaluations queries, the geographical predicates are prefixed by *geo* (`http://rdf.insee.fr/def/geo`) and the demographical predicates are prefixed by *demo* (`http://rdf.insee.fr/def/demo#`). These queries are variations around listing the population count in diverse sub-geographical areas.

**Query $Q_{SELECT}$:**

```
SELECT ?name ?totalPop WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

**Query $Q_{UNION}$:**

```
SELECT ?district ?totalPop WHERE {
 { ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:nom ?name .
   ?dpt geo:subdivisionDirecte ?district .
   FILTER (?v <= "42") }
 UNION {
   ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:nom ?name .
   ?dpt geo:subdivisionDirecte ?district .
   FILTER (?v > "42") }
 ?district demo:population ?popLeg .
```

[4]INSEE demographic data set: `http://rdf.insee.fr/demo/popleg-2013-sc.ttl.zip`

Table 1: Data sets partitioning

| Datasets | Predicates | File size (.ttl) | Number of Triples |
|---|---|---|---|
| $S_1$: Demographic dataset | population , populationTotale | 17,9 Mo | 222 429 |
| $S_3$: Geographic dataset | codeRegion, subdivisionDirecte, nom | 19,9 Mo | 368 761 |
| $S_3$: Geographic dataset copy | codeRegion, subdivisionDirecte, nom | 19,9 Mo | 368 761 |
| $S_4$: Geographic dataset part 1 | codeRegion, subdivisionDirecte, nom | 19,2 Mo | 351 720 |
| $S_5$: Geographic dataset part 2 | codeRegion, subdivisionDirecte, nom | 749,4 ko | 17 217 |
| $S_6$: Geographic dataset part 2.1 | codeRegion, subdivisionDirecte | 735,3 ko | 16 963 |
| $S_7$: Geographic dataset part 2.2 | codeRegion, nom | 15,5 ko | 232 |
| $S_8$: Geographic dataset part 2.3 | subdivisionDirecte, nom | 33,9 ko | 567 |

```
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

**Query $Q_{MINUS}$:**

```
SELECT ?name ?totalPop WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt geo:subdivisionDirecte ?district .
 MINUS {
  ?region geo:codeRegion "24" .
  ?dpt geo:subdivisionDirecte
    <http://id.insee.fr/geo/arrondissement/751> }
 ?district demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

**Query $Q_{FILTER}$:**

```
SELECT ?district ?cantonNom WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt geo:subdivisionDirecte ?district .
 ?district geo:subdivisionDirecte ?canton .
 ?canton geo:nom ?cantonNom .
 FILTER (?v = "11")
 FILTER (?cantonNom = "Paris␣14e␣␣canton")
} ORDER BY ?totalPop
```

**Query $Q_{OPT}$:**

```
SELECT * WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 OPTIONAL {
  ?dpt geo:subdivisionDirecte ?district }
}
```

**Query $Q_{ALL}$:**

```
SELECT ?name ?totalPop WHERE {
 { ?region geo:codeRegion  "24".
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:subdivisionDirecte ?district .
   OPTIONAL { ?district geo:nom ?name }
 } UNION {
   ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:subdivisionDirecte ?district .
   ?district geo:nom ?name .
   MINUS {
    ?dpt geo:subdivisionDirecte
     <http://id.insee.fr/geo/arrondissement/751> .
    ?district geo:subdivisionDirecte
     <http://id.insee.fr/geo/canton/6448> .
    FILTER (?v = "24")
   }
 }
 ?district demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Several metrics are reported for each experiment to measure computing efficiency, both in term of execution time and amount of queries generated:

- Execution time: total execution time measured on the DQP server.
- Processing time: query execution time, excluding the query analysis and rewriting times.
- # sub-queries: number of sub-queries sent to endpoints for evaluation.

In each run, our hybrid strategy (Hybrid), that implements BGP-based evaluation of both vertical and horizontal data partitions, is compared to the reference KGRAM implementation (Reference), that implements a triple-based evaluation strategy with BGP generation for vertical partitions only, similarly to other state-of-the-art KGRAM DQP engines. The correctness of the hybrid algorithm is checked first as all hybrid request produce exactly the same results as their reference counter-part.

## 5.2 Performance and complexity results

Figure 2 displays the performance (execution and processing time) and workload (number of sub-queries generated) for each test query and each partitioning scheme. The execution times shown in Figure 2 (top) are averaged execution time and error bars represent ± 1 standard deviation. The Figure displays groups of 12 measurements for each type of query, shown in the following order: simple SELECT, UNION, MINUS, FILTER, OPTIONAL, and all combined. Each group of 12 measurements is composed of 6 first measurements related to Execution time and 6 last measurements related to Processing time. Each group of 6 measurements shows the Reference and the Hybrid implementations execution time for partitioning schemes $P_1$ (duplication), $P_2$ (global distribution) and $P_3$ (partial distribution). Similarly, Figure 2 (bottom) shows the number of sub-queries processed for each test query. Measurements are further analyzed by partitioning scheme.

As can be seen, the hybrid strategy is consistently faster that the reference strategy in all cases, with execution time reduced by 1.9% to 76% depending on the test case. The number of sub-queries processed is consistently reduced.

The impact of the hybrid approach depends on the efficiency of BGP-based evaluation which, in turn, depends on the distribution of data queried. Thus, the more $L \bowtie$

retrieved results compared to $D\bowtie$, the more efficient the hybrid approach.

In the $P_1$ partitioning scheme, all data is duplicated in two sources. Local joins ($L\bowtie$) will therefore retrieve all the final results. However, the distributed joins ($D\bowtie$) processing is initiated to handle the possible distributed triples and finally deleted by the pruning algorithm to tackle the distributed redundancy issue. Even with this unnecessary processing time, the hybrid approach reduced the processing time by 33% to 53% and the number of sub-queries by 41% to 97% depending on the test query.

In the $P_2$ partitioning scheme, there is no data duplication, and data predicates are globally distributed. More results are retrieved through distributed joins processing. Execution times are consequently higher than in the previous case. The experiments still show a reduction of the processing time by 2.1% to 20% and a reduction of the number of sub-queries by 19% to 48%.

Owing to partial distribution, the $P_3$ partitioning scheme exhibits less selective expressions (partial BGPs) than in the $P_2$ case (global BGP), with a higher chance of matching more results. This explains why the results of the hybrid approach with $P_3$ is more efficient than the results with $P_2$. Furthermore, since the distributed joins are less selective in this case, the pruning algorithm is applied earlier than in the $P_1$ case, which explains why results are improved. In this case, the hybrid approach reduces the processing time by 31% to 80% and the number of sub-queries by 41% to 97%.

# 6. CONCLUSIONS

In this paper, we proposed a query semantics for SPARQL queries applied to distributed linked RDF data set and a hybrid query evaluation algorithm to improve performance of DQP queries. Queries are interpreted as if they were ran on a single virtual RDF graph defined as the set union of all target data sets. Replicated data are consistently accounted only once in the query evaluation, named graph sharing the same identifier are considered as identical and blank nodes are considered unique in the complete data set. The query evaluation algorithm proposed decomposes the query into BGP- and Triple Pattern-based SPARQL sub-queries with filters. It aims at maximising the amount of processing handled by the remote endpoints and minimising the amount of data transferred to the DQP server. The strategy implemented delivers improved results in all kinds of data partitioning schemes and all types of SPARQL queries investigated.

Other query optimisation techniques can be considered to further improve the performance of the KGRAM-DQP engine, in particular better source selection and improved sub-queries ordering heuristics. Another important milestone in the future will be to test the KGRAM-DQP engine in a networked environment to assess the impact of network communications in the query process. KGRAM-DQP also needs to be tested against established benchmarks, such as FedBench [14], and compared to existing competitors for query expressiveness and performance.

# 7. ACKNOWLEDGMENTS

This work was partly funded by the French Government (ANR National Research Agency) through the "Investments for the Future" Program, reference #ANR-11-LABX-0031-01 (Labex UCN@Sophia).

# 8. REFERENCES

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. In *International Conference on The Semantic Web (ISWC'10)*, pages 18–34, Berlin, Heidelberg, 2011.

[2] K. Alexander and M. Hausenblas. Describing linked datasets - on the design and usage of void, the vocabulary of interlinked datasets. In *Linked Data on the Web Workshop (LDOW 09)*, Madrid, Spain, Apr. 2009.

[3] M. Atre. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD Conference*, pages 1793–1808. ACM, 2015.

[4] O. Corby and C. Faron-Zucker. The KGRAM Abstract Machine for Knowledge Graph Querying. In *IEEE/WIC/ACM Web Intelligence(WI 2010)*, , Toronto, Canada, 2010.

[5] O. Corby, A. Gaignard, C. Faron-Zucker, and J. Montagnat. KGRAM Versatile Inference and Query Engine for the Web of Linked Data. In *IEEE/WIC/ACM International Conference on Web Intelligence(WI'12)*, Macao, China, Dec. 2012.

[6] S. H. Garlik, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. 2011.

[7] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Conference on Consuming Linked Data - Volume 782*, COLD'11, pages 13–24, Aachen, Germany, 2010.

[8] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 276–285, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[9] O. Hartig and M. Özsu. Linked data query processing. In *International Conference on Data Engineering (ICDE)*, pages 1286–1289, Chicago, USA, 2014.

[10] T. M. Özsu and P. Valduriez. *Principles of Distributed Database Systems, third edition*. Springer, 2011.

[11] P. Peng, L. Zou, M. Özsu, L. Chen, and D. Zhao. Processing SPARQL queries over distributed RDF graphs. 25(2):243–268, Feb. 2016.

[12] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), Aug. 2009.

[13] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 524–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: optimization techniques for federated query processing on linked data. In *international Semantic Web conference (ISWC'11)*, , pages 601–616, Bonn, Germany, Oct. 2011.

**Figure 2: Top: query processing time. Bottom: sub-queries generated**